Probability Analysis of Monopoly

L. Rotgers (larsrotgers@gmail.com)

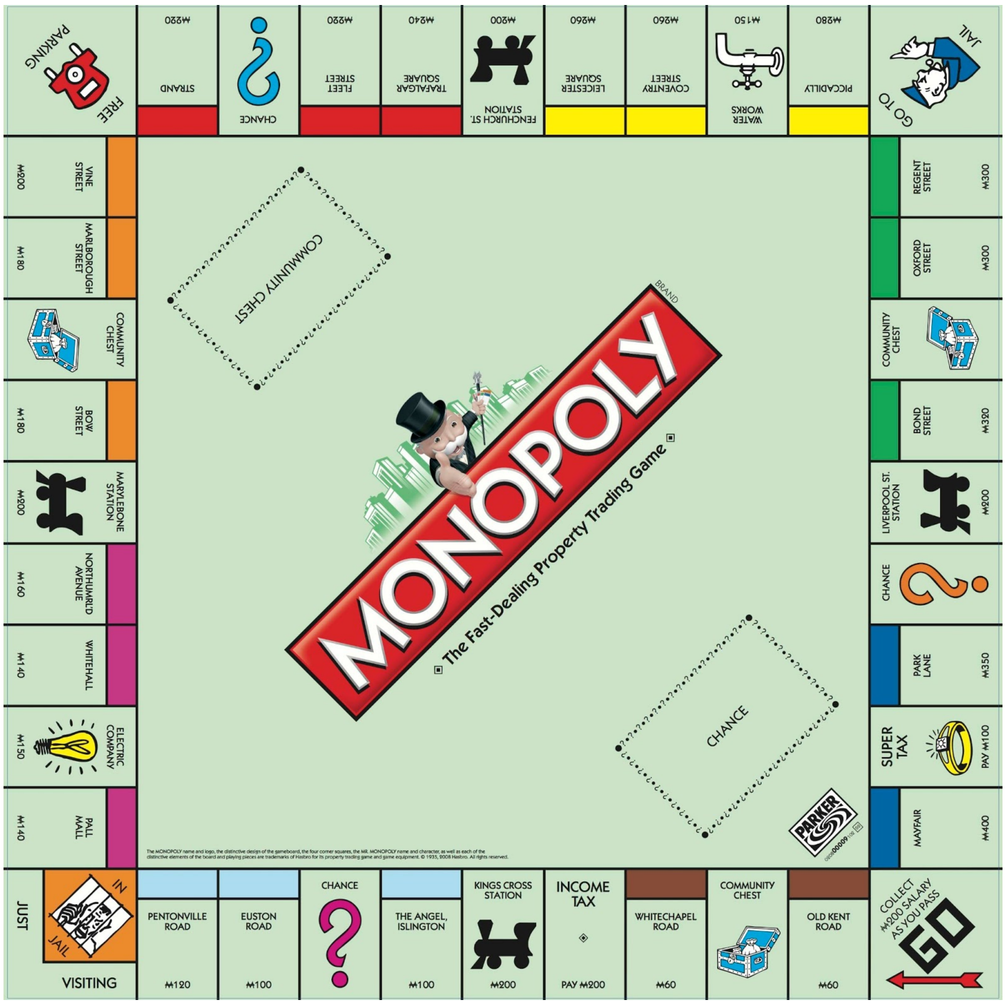
May 13, 2018

Contents

1 Probability Analysis of Monopoly

In this document we are going to analyze the probabilities in Monopoly to answer the question: which are the best houses to buy?

To answer this question we will create a simulated version of Monopoly and determine the probabilities to land on each square. Then we calculate the expected value for each square. The squares with the highest expected values are the best squares to have.



monopoly

1.1 Squares

Each edge has 9 positions and there are 4 edges. There are 4 corners. Which gives a total of 40 positions where a player can land. The labels are numbered starting at GO.

1.1.1 Labels

```
In [2]: squares_labels = ['start', 'b1', 'cc1', 'b2', 'it', 't1',
                          'lb1', 'c1', 'lb2', 'lb3', 'jail', 'p1',
                          'ec', 'p2', 'p3', 'ts2', 'o1', 'cc2',
```

```
                    'o2', 'o3', 'p', 'r1', 'c2', 'r2', 'r3',
                    'ts3', 'y1', 'y2', 'ww', 'y3', 'gtj',
                    'g1', 'g2', 'cc3', 'g3', 'ts4', 'c3',
                    'db1', 'st', 'db2']

squares_total = len(squares_labels)
print('There are {} squares.'.format(squares_total))
```

```
There are 40 squares.
```

1.1.2 Descriptions

We also want to know the proper names, so we don't have to look up the labels.

```
In [3]: squares_description = ['Start', 'Brown 1', 'Community Chest 1', 'Brown 2',
                               'Income Tax', 'Train Station 1', 'Light Blue 1',
                               'Chance 1', 'Light Blue 2', 'Light Blue 3', 'Jail',
                               'Purple 1', 'Electric Company', 'Purple 2',
                               'Purple 3', 'Train Station 2', 'Orange 1',
                               'Community Chest 2', 'Orange 2', 'Orange 3',
                               'Free Parking', 'Red 1', 'Chance 2', 'Red 2',
                               'Red 3', 'Train Station 3', 'Yellow 1', 'Yellow 2',
                               'Water Works', 'Yellow 3', 'Go to Jail', 'Green 1',
                               'Green 2', 'Community Chest 3', 'Green 3',
                               'Train Station 4', 'Chance 3', 'Dark Blue 1',
                               'Super Tax', 'Dark Blue 2']
```

```
In [4]: print('There are {} descriptions.'.format(len(squares_description)))
```

```
There are 40 descriptions.
```

1.1.3 Purchasable

We want to know if they are purchasable so we can sort on that later.

```
In [5]: squares_purchasable = [False, True, False, True, False,
                               True, True, False, True, True,
                               False, True, True, True, True,
                               True, True, False, True, True,
                               False, True, False, True, True,
                               True, True, True, True, True,
                               False, True, True, False, True,
                               True, False, True, False, True]
```

1.1.4 Rent

We want to use the rent paid at each square to calculate the expected value. The utility company charge 4 times roll if one is owned, and 10 times roll if both owned. For one railway we charge 25, two 50, three 100, and all four 200.

To find the rent for a utility company, we find the expected value for throwing a dices times 4.

$$4 \cdot E(\bar{k}) = 4 \cdot \frac{1}{6} \cdot (1 + 2 + 3 + 4 + 5 + 6)$$

```
In [6]: E_u = 4 * (1+2+3+4+5+6) / 6
```

We pick the value for one railway which is 25.

```
In [7]: E_r = 25
```

```
In [8]: squares_rent = [0, 2, 0, 4, 0, E_r, 6, 0, 6, 8, 0, 10, E_u,
                        10, 12, E_r, 14, 0, 14, 16, 0, 18, 0, 18,
                        20, E_r, 22, 22, E_u, 24, 0, 26, 26, 0, 28,
                        E_r, 0, 35, 0, 50]
```

1.1.5 Grouping

We want to know in what group they are so we can aggregate our data later.

```
In [9]: squares_aggregate = ['Start', 'Brown', 'Community Chest', 'Brown',
                             'Income Tax', 'Train Station', 'Light Blue',
                             'Chance', 'Light Blue', 'Light Blue', 'Jail',
                             'Purple', 'Utilities', 'Purple', 'Purple',
                             'Train Station', 'Orange', 'Community Chest',
                             'Orange', 'Orange', 'Free Parking', 'Red',
                             'Chance', 'Red', 'Red', 'Train Station', 'Yellow',
                             'Yellow', 'Utilities', 'Yellow', 'Go to Jail',
                             'Green', 'Green', 'Community Chest', 'Green',
                             'Train Station', 'Chance', 'Dark Blue',
                             'Super Tax', 'Dark Blue']
```

1.2 Cards

There are two decks of cards.

- Community Cards
- Chance Cards

Each deck contains 16 cards.

1.2.1 Community Cards

Monopoly has 16 community cards.

cc

Because we are only determining the probabilities, we are only interested in the following cards:

- advance to go
- go to jail
- get out of jail, free
- go back 2 spaces

1.2.2 Community deck implementation

We implement the community deck in a class. The class keeps track of a list with 16 cards. An index points to the next card. When we are out of cards, we reset the index and reshuffle the cards.

```
In [10]: from random import shuffle

         class CommunityDeck():
             def __init__(self):
                 self.deck = [0] * 16
                 self.deck[0] = 'gtg' # go to go
                 self.deck[1] = 'gtj' # go to jail
                 self.deck[2] = 'goj' # get out of jail
                 self.deck[3] = 'gb2' # go back 2 steps
                 self.index = 16

             def draw_card(self):
                 if self.index >= len(self.deck):
```

```
            self.index = 0
            shuffle(self.deck)
        card = self.deck[self.index]
        self.index += 1
        return card
```

Now we test it:

```
In [11]: deck = CommunityDeck()
         deck.deck
```

```
Out[11]: ['gtg', 'gtj', 'goj', 'gb2', 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

```
In [12]: deck.draw_card()
```

```
Out[12]: 0
```

```
In [13]: deck.deck
```

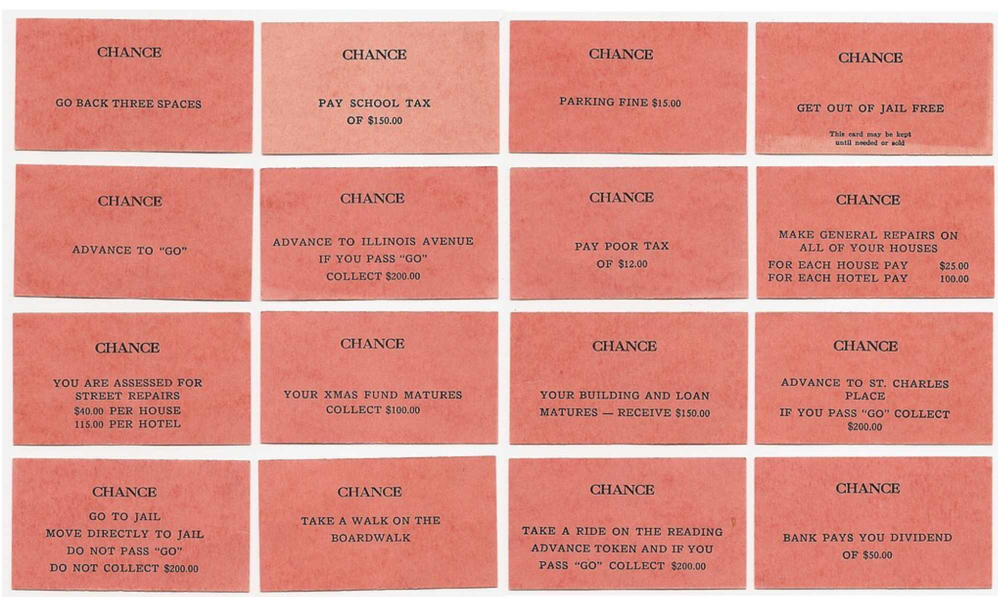
```
Out[13]: [0, 0, 0, 0, 'gtj', 0, 0, 0, 'gb2', 'gtg', 'goj', 0, 0, 0, 0, 0]
```

```
In [14]: deck.index
```

```
Out[14]: 1
```

1.2.3 Chance Cards

Monopoly has 16 chance cards.



chance

Because we are only determining the probabilities, we are only interested in the following cards:

- go back three spaces
- get out of jail free
- advance to go
- advance to illinois avenue (R3)
- go to jail

1.2.4 Chance deck implementation

We implement the chance deck in a class. The class keeps track of a list with 16 cards. An index points to the next card. When we are out of cards, we reset the index and reshuffle the cards.

```
In [15]: from random import shuffle

         class ChanceDeck():
             def __init__(self):
                 self.deck = [0] * 16
                 self.deck[0] = 'gtg' # go to go
                 self.deck[1] = 'gtj' # go to jail
                 self.deck[2] = 'goj' # get out of jail
                 self.deck[3] = 'gb3' # go back 3
                 self.deck[4] = 'r3'  # go to red 3 (r3)
                 self.index = 16

             def draw_card(self):
                 if self.index >= len(self.deck):
                     self.index = 0
                     shuffle(self.deck)
                 card = self.deck[self.index]
                 self.index += 1
                 return card
```

Now we test it:

```
In [16]: deck = ChanceDeck()
         deck.deck

Out[16]: ['gtg', 'gtj', 'goj', 'gb3', 'r3', 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]

In [17]: deck.draw_card()

Out[17]: 0

In [18]: deck.deck

Out[18]: [0, 'gtj', 'gtg', 0, 0, 0, 'gb3', 0, 0, 0, 0, 0, 'r3', 0, 0, 'goj']

In [19]: deck.index

Out[19]: 1
```

1.3 Dice

We will be implementing the dice as a class. This allows us to encapsulate how the result are determined. It makes it easier to implements other scenarios such as throwing with multiple dices.

```
In [20]: from random import randint

         class Dice():
             def __init__(self, dices = 1, sides = 6):
                 self.dices = dices
                 self.sides = 6

             def throw(self):
                 return sum([randint(1, self.sides) for _ in range(self.dices)])
```

Rolling one time:

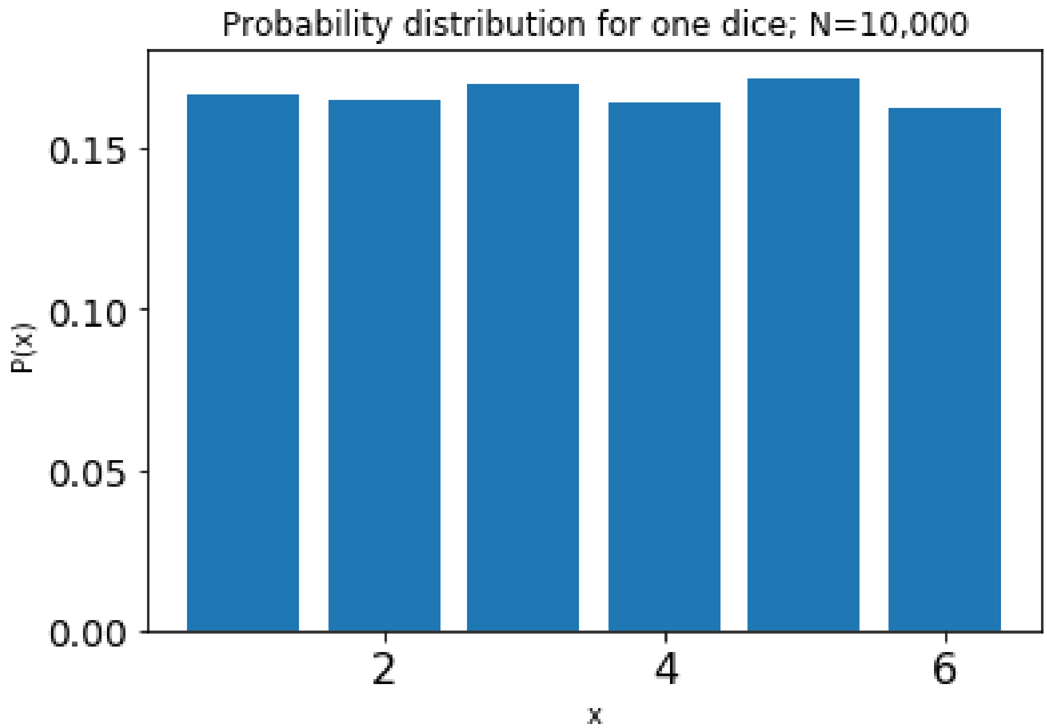
```
In [21]: dice = Dice()
         dice.throw()

Out[21]: 1
```

1.3.1 Simple: one dice with six sides

A simple setup would be one dice with six sides. This will give uniformly distributed probabilities.

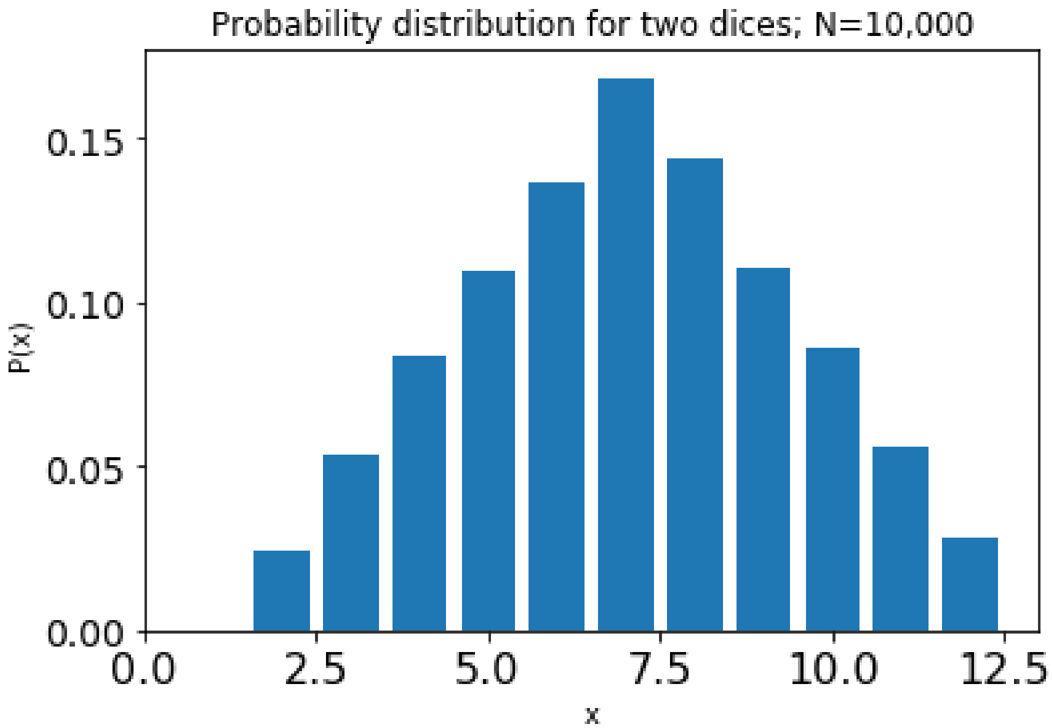
```
In [88]: dice = Dice()
         sides = [0] * dice.dices * dice.sides
         N = 10000
         for i in range(N): sides[dice.throw()-1] += 1
         sides = np.array(sides) / N
         bar(range(1,len(sides)+1), sides);
         ylabel('P(x)')
         xlabel('x')
         title('Probability distribution for one dice; N=10,000');
```



1.3.2 Advanced: two dice with six sides

Monopoly is played with two dices. This will the following probability distribution:

```
In [89]: dice = Dice(2)
         sides = [0] * dice.dices * dice.sides
         N = 10000
         for i in range(N): sides[dice.throw()-1] += 1
         sides = np.array(sides) / N
         bar(range(1,len(sides)+1), sides);
         ylabel('P(x)')
         xlabel('x')
         title('Probability distribution for two dices; N=10,000');
```



1.4 Monopoly simulation

We are playing a simplified version of Monopoly. We will not keep track of money. There will only be one player. If a player goes to jail, the player can continue immediately on the next turn. We will keep track of the card decks. The game will be played with 2 dices. We only count when we land on a square. If we are moved to jail for example, the next round will continue from that new position.

1.4.1 Algorithm

Here we are going to simulate a game for N amount of rounds. The game algorithm is simple:

1. Roll the dices (there are 2 dices with 6 squares)
2. Move to the new position
3. Increment the square counter for that position
4. Check and handle go to jail
5. Check and handle community chest
6. Check and handle chance

1.4.2 Modulo arithmetic for position tracking

We can easily keep track of our position with modulo arithemetic. Let C be our position (or `index`), d the result from throwing the dice, and n the current round. To determine our new position we calculate:

$$C_{n+1} \equiv C_n + d \pmod{40}$$

The modulo is 40 because that are the total amount of squares.

1.4.3 Implementation

Below is the implementation for the Monopoly simulation.

```
In [90]: dice = Dice(2)
         community_deck = CommunityDeck()
         chance_deck = ChanceDeck()

         index = 0 # position
         total_squares = len(squares_labels)
         squares = [0] * total_squares

         rounds = 1000000 # N

         for i in range(rounds):

             # Throw the dice and move our position on the board.
             steps = dice.throw()
             index = (index + steps) % total_squares
             squares[index] += 1

             # We landed on go to jail.
             if squares_labels[index] is 'gtj':
                 index = squares_labels.index('jail')

             # We landed on the community card.
             if squares_labels[index] in ['cc1', 'cc2', 'cc3']:
                 card = community_deck.draw_card()
                 if card is 'gtg': index = squares_labels.index('start')
                 if card is 'gtj': index = squares_labels.index('jail')
                 if card is 'gb2':
                     if index >= 2: index -= 2
                     if index < 2: index = total_squares-abs(index-2)-1

             # We landed on the chance card.
             if squares_labels[index] in ['c1', 'c2', 'c3']:
                 card = chance_deck.draw_card()
                 if card is 'gtg': index = squares_labels.index('start')
                 if card is 'gtj': index = squares_labels.index('jail')
                 if card is 'r3': index = squares_labels.index('r3')
                 if card is 'gb3':
                     if index >= 3: index -= 3
                     if index < 3: index = total_squares-abs(index-3)-1
```

It takes around 2.7 seconds to run a game when $N = 1,000,000$. Because there is only one loop the algorithm will scale linearly.

1.5 Probability statistics

Now we can proceed to analyze our results.

1.5.1 Determining probabilities

With the number of times that each square is visited we can calculate the probabilities. The probability that a square is visited is:

$$P(\bar{x} = x) = \frac{\text{Times visited}}{\text{\# of rounds}}$$

We can calculate the expected value for each square in terms of money with:

$$E(\bar{k}) = P(\bar{x} = x) \cdot \text{Rent}$$

We also want to create a `DataFrame` in Python to easily keep track of everything.

```
In [25]: import pandas as pd
         df = pd.DataFrame(index=range(total_squares))
         df['Square'] = squares_labels
         df['Description'] = squares_description
         df['Purchasable'] = squares_purchasable
         df['Rent'] = squares_rent
         df['Visited'] = squares
         df['Probability'] = df['Visited'] / rounds
         df['Aggregate'] = squares_aggregate
         df['Expected Value'] = df['Probability'] * df['Rent']
```

We can calculate a quick summary about the data:

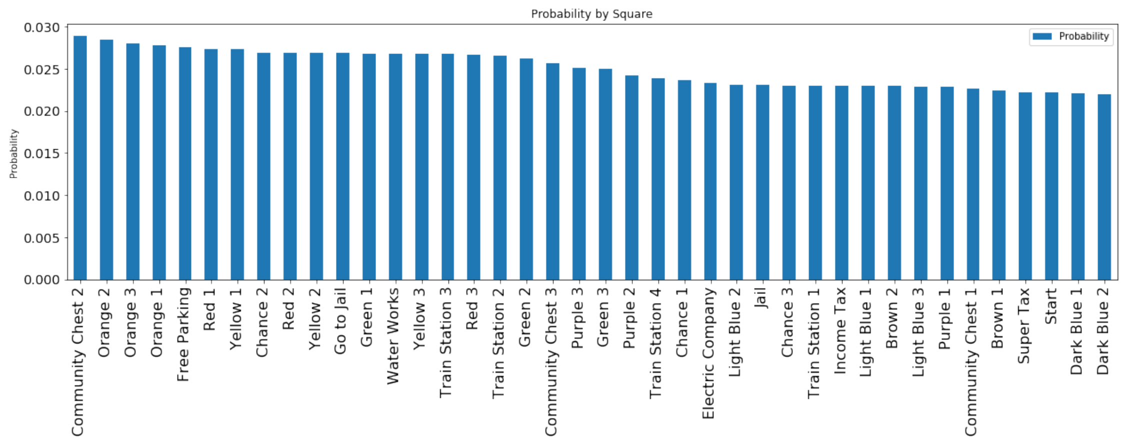
```
In [26]: print('Total rounds: {}'.format(rounds))
         print('Visited avg:  {}'.format(df['Visited'].mean()))
         print('Visited min:  {}'.format(df['Visited'].min()))
         print('Visited max:  {}'.format(df['Visited'].max()))
         print('Visited std:  {:.2f}'.format(df['Visited'].std()))
```

```
Total rounds: 1000000
Visited avg:  25000.0
Visited min:  21973
Visited max:  28895
Visited std:  2175.56
```

1.5.2 Plot of probabilities by square

If we sort these values descending on the probability, we can easily see which squares have the highest probability to be visited.

```
In [27]: plt.rc('xtick', labelsize=16)
         plt.rc('ytick', labelsize=14)
         df[['Description', 'Probability']].sort_values(by='Probability', ascending=False)\
             .plot(kind='bar', figsize=(20,5))
         plt.xticks(range(total_squares), df[['Description', 'Probability']]
             .sort_values(by='Probability', ascending=False)['Description'])
         plt.ylabel('Probability')
         plt.title('Probability by Square');
```



Here we can conclude that `Orange 1` is the most visited square. Also notice that `Orange 2` and `Orange 3` are pretty high. It seems that `Orange` is the best street to have.

1.5.3 Table of probabilities by square

Below is the full table with all the squares and their corresponding values.

```
In [102]: df.loc[:, df.columns.isin(['Square', 'Description', 'Probability'])].sort_values(by='P
```

```
Out[102]:     Square         Description  Probability
          17     cc2  Community Chest 2     0.028895
          18      o2           Orange 2     0.028447
          19      o3           Orange 3     0.028035
          16      o1           Orange 1     0.027736
          20       p       Free Parking     0.027539
          21      r1              Red 1     0.027316
          26      y1           Yellow 1     0.027288
          22      c2           Chance 2     0.026892
          23      r2              Red 2     0.026878
          27      y2           Yellow 2     0.026876
```

30	gtj	Go to Jail	0.026837
31	g1	Green 1	0.026799
28	ww	Water Works	0.026769
29	y3	Yellow 3	0.026719
25	ts3	Train Station 3	0.026718
24	r3	Red 3	0.026641
15	ts2	Train Station 2	0.026571
32	g2	Green 2	0.026236
33	cc3	Community Chest 3	0.025676
14	p3	Purple 3	0.025082
34	g3	Green 3	0.024969
13	p2	Purple 2	0.024221
35	ts4	Train Station 4	0.023904
7	c1	Chance 1	0.023619
12	ec	Electric Company	0.023254
8	lb2	Light Blue 2	0.023096
10	jail	Jail	0.023040
36	c3	Chance 3	0.023013
5	t1	Train Station 1	0.023001
4	it	Income Tax	0.022989
6	lb1	Light Blue 1	0.022955
3	b2	Brown 2	0.022941
9	lb3	Light Blue 3	0.022853
11	p1	Purple 1	0.022802
2	cc1	Community Chest 1	0.022639
1	b1	Brown 1	0.022416
38	st	Super Tax	0.022214
0	start	Start	0.022133
37	db1	Dark Blue 1	0.022018
39	db2	Dark Blue 2	0.021973

1.5.4 Top 10 highest probability squares

The top 10 squares that have the highest probability for a player to land on are:

```
In [101]: df.loc[df['Purchasable'] == True, df.columns.isin(['Square', 'Description', 'Probabili
              .sort_values('Probability', ascending=False).head(10)
```

Out[101]:

	Square	Description	Probability
18	o2	Orange 2	0.028447
19	o3	Orange 3	0.028035
16	o1	Orange 1	0.027736
21	r1	Red 1	0.027316
26	y1	Yellow 1	0.027288
23	r2	Red 2	0.026878
27	y2	Yellow 2	0.026876
31	g1	Green 1	0.026799
28	ww	Water Works	0.026769
29	y3	Yellow 3	0.026719

The total probability for all 10 squares is:

```
In [30]: df.loc[df['Purchasable'] == True].sort_values('Probability', ascending=False)\
             .head(10)['Probability'].sum()
```

```
Out[30]: 0.27286299999999997
```

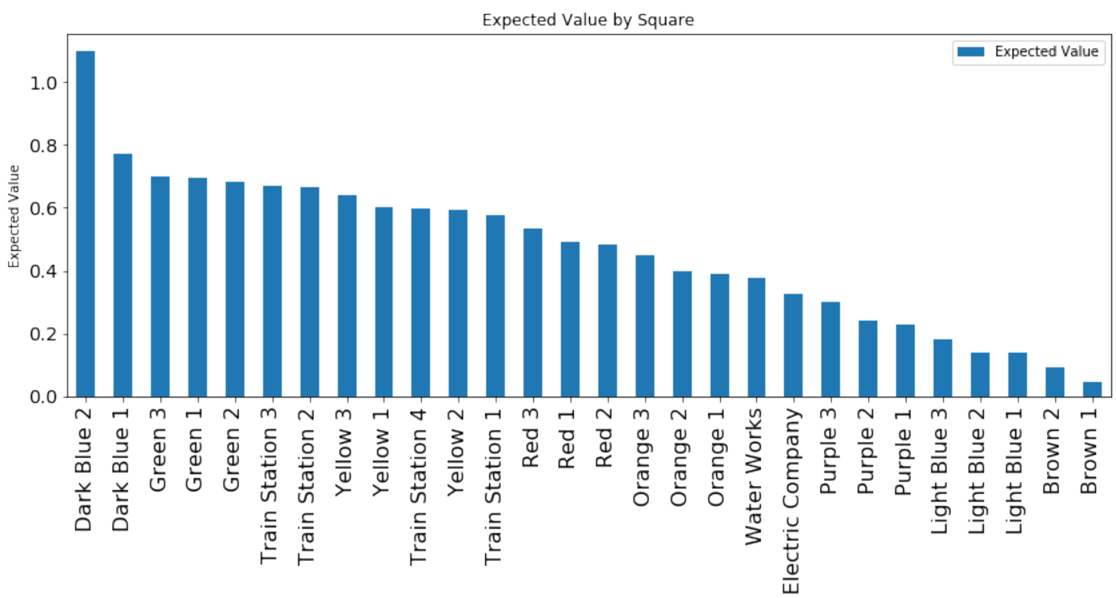
1.5.5 Plot of expected value per turn

Now we want to know how much each square generates per turn with the found probabilities and the rent the player needs to pay when we land on it. First we make a selection to only get the purchasable squares.

```
In [31]: rentable_df = pd.DataFrame(df.loc[df['Purchasable'] == True]).reindex()
```

Plotting this gives:

```
In [32]: plt.rc('xtick', labelsize=16)
         plt.rc('ytick', labelsize=14)
         rentable_df[['Description', 'Expected Value']].sort_values(by='Expected Value', ascendi
             .plot(kind='bar', figsize=(14,5))
         plt.xticks(range(len(rentable_df.index)), rentable_df[['Description', 'Expected Value']
             .sort_values(by='Expected Value', ascending=False)['Description'])
         plt.ylabel('Expected Value')
         plt.title('Expected Value by Square');
```



1.5.6 Table of expected value per turn

The full table of expected values is below.

```
In [116]: rentable_df.loc[:, rentable_df.columns.isin(['Square', 'Description', 'Rent', 'Probabi
              .sort_values('Expected Value', ascending=False)
```

Out[116]:

	Square	Description	Rent	Probability	Expected Value
39	db2	Dark Blue 2	50.0	0.021973	1.098650
37	db1	Dark Blue 1	35.0	0.022018	0.770630
34	g3	Green 3	28.0	0.024969	0.699132
31	g1	Green 1	26.0	0.026799	0.696774
32	g2	Green 2	26.0	0.026236	0.682136
25	ts3	Train Station 3	25.0	0.026718	0.667950
15	ts2	Train Station 2	25.0	0.026571	0.664275
29	y3	Yellow 3	24.0	0.026719	0.641256
26	y1	Yellow 1	22.0	0.027288	0.600336
35	ts4	Train Station 4	25.0	0.023904	0.597600
27	y2	Yellow 2	22.0	0.026876	0.591272
5	t1	Train Station 1	25.0	0.023001	0.575025
24	r3	Red 3	20.0	0.026641	0.532820
21	r1	Red 1	18.0	0.027316	0.491688
23	r2	Red 2	18.0	0.026878	0.483804
19	o3	Orange 3	16.0	0.028035	0.448560
18	o2	Orange 2	14.0	0.028447	0.398258
16	o1	Orange 1	14.0	0.027736	0.388304
28	ww	Water Works	14.0	0.026769	0.374766
12	ec	Electric Company	14.0	0.023254	0.325556
14	p3	Purple 3	12.0	0.025082	0.300984
13	p2	Purple 2	10.0	0.024221	0.242210
11	p1	Purple 1	10.0	0.022802	0.228020
9	lb3	Light Blue 3	8.0	0.022853	0.182824
8	lb2	Light Blue 2	6.0	0.023096	0.138576
6	lb1	Light Blue 1	6.0	0.022955	0.137730
3	b2	Brown 2	4.0	0.022941	0.091764
1	b1	Brown 1	2.0	0.022416	0.044832

1.6 Grouped probability statistics

We want to answer the following questions:

1. What are the best streets to have?
2. What is the probability to be in jail?
3. What is the probability to draw a card?

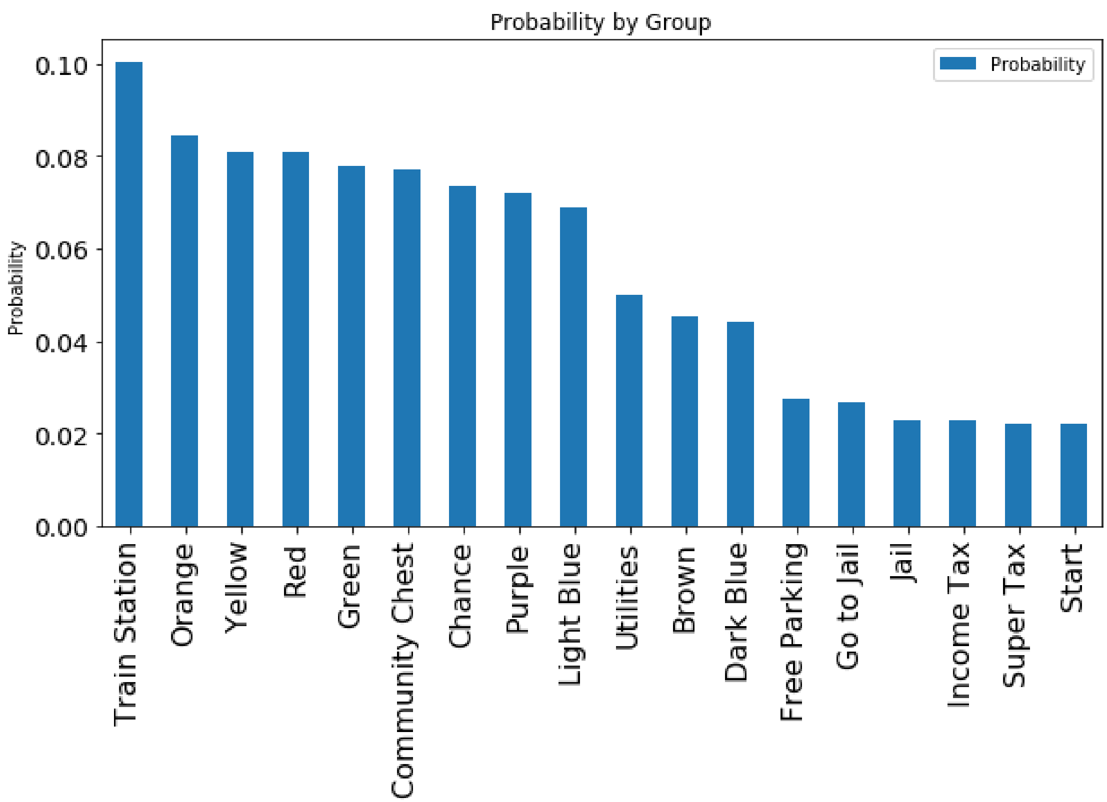
1.6.1 Plot of probabilities by group

To find what the probabilities are per street, chance, community chest, etc., we are going to aggregate the possibilities.

```
In [34]: aggregated_df = pd.DataFrame(df.groupby(['Aggregate'])['Probability'].sum()).reset_inde
```

Now we plot the aggregated probabilities.

```
In [35]: plt.rc('xtick', labelsize=16)
         plt.rc('ytick', labelsize=14)
         aggregated_df[['Aggregate', 'Probability']].sort_values(by='Probability', ascending=Fal
             .plot(kind='bar', figsize=(10,5))
         plt.xticks(range(len(aggregated_df.index)), aggregated_df[['Aggregate', 'Probability']]
             .sort_values(by='Probability', ascending=False)['Aggregate'])
         plt.ylabel('Probability')
         plt.title('Probability by Group');
```



1.6.2 Table of probabilities by group

A total overview of all the probabilities can be found in the table below:

```
In [36]: aggregated_df.sort_values('Probability', ascending=False)
```

```
Out[36]:          Aggregate  Probability
         15   Train Station     0.100194
         10          Orange     0.084218
```

```
17          Yellow     0.080883
12             Red     0.080835
6            Green     0.078004
2  Community Chest     0.077210
1           Chance     0.073524
11          Purple     0.072105
9       Light Blue     0.068904
16       Utilities     0.050023
0            Brown     0.045357
3        Dark Blue     0.043991
4     Free Parking     0.027539
5       Go to Jail     0.026837
8             Jail     0.023040
7       Income Tax     0.022989
14       Super Tax     0.022214
13           Start     0.022133
```

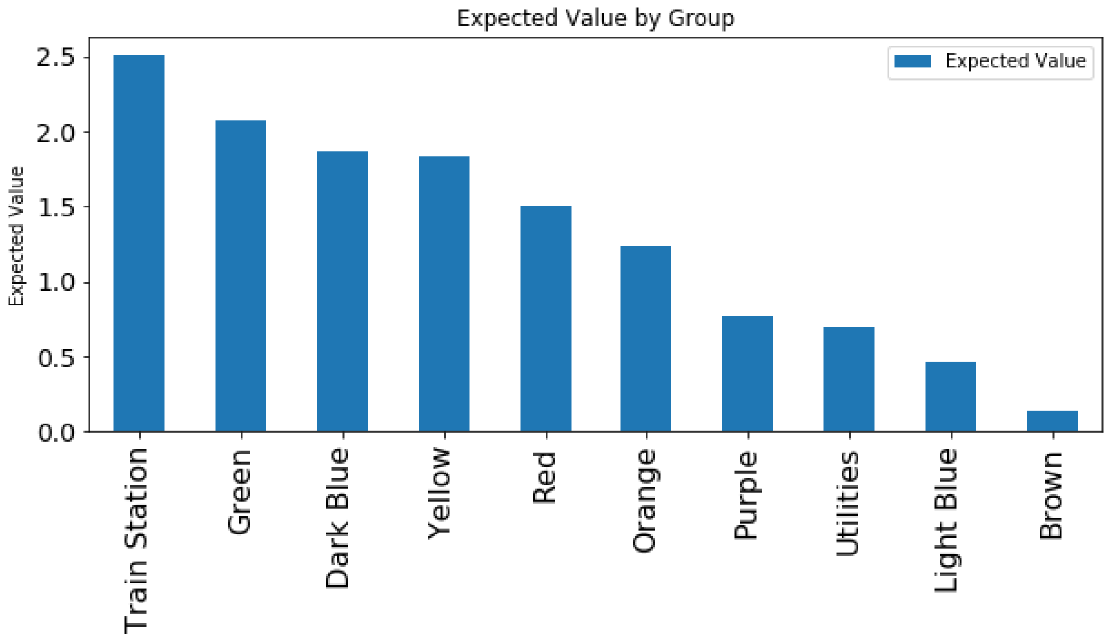
1.6.3 Plot of expected values by group

If we find the expected values by each aggregate we can find out which group generated the most money per turn.

```
In [117]: aggregated_ev_df = pd.DataFrame(df.loc[df['Purchasable'] == True]\
              .groupby(['Aggregate'])['Expected Value'].sum()).reset_index()
```

Plotting this gives:

```
In [38]: plt.rc('xtick', labelsize=16)
         plt.rc('ytick', labelsize=14)
         aggregated_ev_df[['Aggregate', 'Expected Value']].sort_values(by='Expected Value', asce
             .plot(kind='bar', figsize=(10,4))
         plt.xticks(range(len(aggregated_ev_df.index)), aggregated_ev_df[['Aggregate', 'Expected
             .sort_values(by='Expected Value', ascending=False)['Aggregate'])
         plt.ylabel('Expected Value')
         plt.title('Expected Value by Group');
```



We can conclude that the `Train Station` yields the most. This are however 4 squares. The best street to have is `Green`.

1.6.4 Table of expected values by group

A full table of expected values can be found below

```
In [39]: aggregated_ev_df.sort_values('Expected Value', ascending=False)
```

```
Out[39]:      Aggregate  Expected Value
         7  Train Station        2.504850
         2          Green        2.078042
         1      Dark Blue        1.869280
         9         Yellow        1.832864
         6            Red        1.508312
         4         Orange        1.235122
         5         Purple        0.771214
         8      Utilities        0.700322
         3     Light Blue        0.459130
         0          Brown        0.136596
```

1.7 Other probabilities

1.7.1 Train station probabilities

We can conclude that `Train Station` has the highest probability to land on. However, we need to take into account that there are four squares to land on.

```
In [119]: df.loc[df['Aggregate'] == 'Train Station', df.columns.isin(['Square', 'Description', '
              .sort_values('Probability', ascending=False)
```

```
Out[119]:    Square      Description  Probability  Expected Value
          25    ts3  Train Station 3     0.026718        0.667950
          15    ts2  Train Station 2     0.026571        0.664275
          35    ts4  Train Station 4     0.023904        0.597600
          5      t1  Train Station 1     0.023001        0.575025
```

1.7.2 Probability to be in jail

To find the total probability to be in jail, we need to take into account that:

- We can land on jail.
- We can land on go to jail.
- There is one community card which sends you to jail.
- There is one chance card which sends you to jail.

Each deck has 16 cards, therefore the probability to draw go to jail is $P(\bar{x} = \text{go to jail}) = \frac{1}{16}$.

$$P(\bar{x} = \text{in jail}) = P(\bar{x} = \text{jail}) + P(\bar{x} = \text{go to jail}) + \frac{1}{16}\left[P(\bar{x} = \text{community chest}) + P(\bar{x} = \text{chance})\right]$$

```
In [41]: P_jail            = sum(aggregated_df.loc[aggregated_df['Aggregate'] == 'Jail'
         P_go_to_jail      = sum(aggregated_df.loc[aggregated_df['Aggregate'] == 'Go to Jail'
         P_community_card  = sum(aggregated_df.loc[aggregated_df['Aggregate'] == 'Community Chest
         P_chance_card     = sum(aggregated_df.loc[aggregated_df['Aggregate'] == 'Chance'
```

```
In [42]: P_jail + P_go_to_jail + 1/16 * (P_community_card + P_chance_card)
```

```
Out[42]: 0.059297875000000007
```

1.7.3 Probability to draw a card

To find the probability to draw a card, we simply calculate:

$$P(\bar{x} = \text{draw a card}) = P(\bar{x} = \text{community chest}) + P(\bar{x} = \text{chance})$$

```
In [43]: P_community_card + P_chance_card
```

```
Out[43]: 0.15073400000000001
```

Where the probabilities for the community chest square are:

```
In [120]: df.loc[df['Aggregate'] == 'Community Chest', df.columns.isin(['Square', 'Description',
              .sort_values('Probability', ascending=False)
```

```
Out[120]:    Square        Description  Probability
          17    cc2  Community Chest 2     0.028895
          33    cc3  Community Chest 3     0.025676
          2     cc1  Community Chest 1     0.022639
```

And the probabilities for the chance square are:

```
In [121]: df.loc[df['Aggregate'] == 'Chance', df.columns.isin(['Square', 'Description', 'Probabi
              .sort_values('Probability', ascending=False)
```

```
Out[121]:    Square Description  Probability
          22     c2    Chance 2     0.026892
          7      c1    Chance 1     0.023619
          36     c3    Chance 3     0.023013
```

1.8 Conclusion

We saw the expected values per turn for each square. Train stations are by far the best to buy. The multiplier for them is not taken into account here. Which means that the actual results are even better. After that, buying should be done in a priority. I wouldn't buy utilities, brown and light blue squares because they have a very low expected value.

1.8.1 Top 10 streets

A table of the top 10 squares by expected value.

```
In [122]: df.loc[:,df.columns.isin(['Square', 'Description', 'Probability', 'Expected Value'])]\
              .sort_values('Expected Value', ascending=False).head(10)
```

```
Out[122]:    Square      Description  Probability  Expected Value
          39    db2      Dark Blue 2     0.021973        1.098650
          37    db1      Dark Blue 1     0.022018        0.770630
          34     g3          Green 3     0.024969        0.699132
          31     g1          Green 1     0.026799        0.696774
          32     g2          Green 2     0.026236        0.682136
          25    ts3  Train Station 3     0.026718        0.667950
          15    ts2  Train Station 2     0.026571        0.664275
          29     y3         Yellow 3     0.026719        0.641256
          26     y1         Yellow 1     0.027288        0.600336
          35    ts4  Train Station 4     0.023904        0.597600
```

1.8.2 Top 10 groups

A table of the top 10 groups by expected value.

```
In [47]: aggregated_ev_df.sort_values('Expected Value', ascending=False).head(10)
```

```
Out[47]:        Aggregate  Expected Value
         7  Train Station        2.504850
         2          Green        2.078042
         1      Dark Blue        1.869280
         9         Yellow        1.832864
         6            Red        1.508312
         4         Orange        1.235122
```

```
5         Purple       0.771214
8      Utilities       0.700322
3     Light Blue       0.459130
0          Brown       0.136596
```

Additional note about train stations Notice that if you own all 4 train stations, any visitor has to pay $200 instead of the $25 for owning one. From this we can find that the expected value is multiplied by 8, if you own them all.

```
In [61]: 8 * df.loc[df['Aggregate'] == 'Train Station']['Expected Value'].sum()
```

```
Out[61]: 20.038799999999998
```

And for 3 train stations, any visitor has to pay $100, which multiplied by 4 gives:

```
In [62]: 4 * df.loc[df['Aggregate'] == 'Train Station']['Expected Value'].sum()
```

```
Out[62]: 10.019399999999999
```

Finally for 2 train stations, any visitor has to pay $50, which multiplied by 2 gives:

```
In [63]: 2 * df.loc[df['Aggregate'] == 'Train Station']['Expected Value'].sum()
```

```
Out[63]: 5.0096999999999996
```

Now keep in mind that if you own the **entire** best street, the results are:

```
In [78]: aggregated_ev_df.sort_values('Expected Value', ascending=False).iloc[[1,2,3]]
```

```
Out[78]:    Aggregate  Expected Value
        2       Green        2.078042
        1   Dark Blue        1.869280
        9      Yellow        1.832864
```

1.8.3 Probability to go to jail

The probability to go to jail is:

```
In [48]: P_jail + P_go_to_jail + 1/16 * (P_community_card + P_chance_card)
```

```
Out[48]: 0.059297875000000007
```

1.8.4 Probability to draw a card

The probability to draw a card is:

```
In [49]: P_community_card + P_chance_card
```

```
Out[49]: 0.15073400000000001
```